Methodologies for Diagnosis of Unreachable States via Property Directed Reachability

Ryan Berryhill, *Student Member, IEEE*, and Andreas Veneris, *Senior Member, IEEE*

***Abstract*—In the modern design cycle, substantial manual effort is required to correct failed liveness properties due to the limited availability of automated tools. To address this limitation, this paper introduces two techniques to diagnose register transfer level errors that manifest in the form of erroneously unreachable states, which represent a common form of liveness property failure. The first uses steps of reachable state-space over-approximation and traditional debugging to compute a subset of the solutions that make a target state reachable. The second solves a series of unbounded model checking problems using an enhanced model of the circuit's transition relation to compute the complete solution set to the problem. The proposed techniques are complementary to each other and present the user with a configurable tradeoff between runtime and resolution of the returned solution set. Empirical results on OpenCores and HWMCC'15 circuits confirm the effectiveness of the approaches and demonstrate the tradeoffs between them.**

***Index Terms*—Debugging, formal verification, IC3, property directed reachability (PDR), register transfer level (RTL), verification.**

I. Introduction

THE PRIMARY challenge in modern very large scale integration design is verification, which accounts for upward of 50% of the design effort [1]. Design debugging, which involves localizing and correcting a failure after it is revealed by verification, accounts for half of this effort. The constant increase in design complexity further complicates these processes, increasing costs, and time-to-market. Formal verification techniques [2]–[5] have grown more scalable with recent advances [6]–[9] making them increasingly applicable to the challenge of verifying modern designs. Likewise, automated debugging tools [10] have seen numerous advancements targeting scalability toward the goal of tackling modern debugging problems [11]–[14].

Functional verification involves simulation or model checking to ensure that specific properties hold on a design. Properties can be divided into two broad classes: 1) safety properties and 2) liveness properties [15]. When a safety property is found to fail through means, such as simulation and model checking, a finite counter-example demonstrating the failure is returned as a certificate. The counter-example forms an error-trace that can be used with an automated debugging tool [10], [16] to aid the engineer in finding the error. Evidently, a great deal of automation is available to complement the human effort to find the root cause of a failure when error-traces are available.

In contrast, a liveness property has no finite counter-examples [15]. A commonly used type of liveness property holds when a particular state is reachable. Such a property can be verified using safety checking techniques [2] or exhaustive simulation. Since it is a liveness property, no error-traces are available upon failure, stifling attempts to apply the aforementioned automated tools. Attempts to correct the failure therefore typically involve a first step in which the engineer attempts to manually discover a trace that should reach the target state but erroneously reaches some other state. This is a largely manual and time-consuming task with little automation available beyond iterative manual processes partially assisted by traditional computer-aided design (CAD) tools [17].

To mitigate this cost, this paper presents two complementary techniques to diagnose this kind of failure in the absence of any error-traces. The first is an approximate approach [18], [19] that returns a subset of all candidate fault sites, where a correction may rectify the failure (i.e., solutions) by making the target state reachable. The solutions returned are dependent on a set of parameters supplied by the user that dictate and limit the portion of the solution space to be explored. The safety checking technique of property directed reachability (PDR) [2], [7] is applied along with boolean satisfiability (SAT)-based debugging [10] to explore the relevant portion of the solution space. The second contribution is an exact approach [19], [20] that returns the complete set of all solutions at the cost of increased runtime when compared to the approximate approach. This approach applies PDR in a novel incremental fashion that substantially reduces runtime by reusing previous results.

In greater detail, the approximate approach returns a subset of all solutions and operates as follows. An incremental PDR solver is executed toward the goal of proving that the target state is unreachable. As a side effect, this computes over-approximations of the reachable state space in each operational design cycle for a bounded number of cycles. This approximation provides a set of constraints that is used with traditional SAT-based debugging to find a set of locations that, when modified, allow the design to transition from a state in the over-approximation to the unreachable target state. Spurious solutions are detected and discarded using the same PDR

Manuscript received September 1, 2016; revised January 10, 2017 and May 26, 2017; accepted July 23, 2017. Date of publication August 31, 2017; date of current version May 18, 2018. This paper was recommended by Associate Editor P. Dasgupta. *(Corresponding author: Ryan Berryhill.)*

The authors are with the University of Toronto, Toronto, ON 5S 3G4, Canada (e-mail: ryan@eecg.toronto.edu).

Digital Object Identifier 10.1109/TCAD.2017.2747999

solver, which has the beneficial side effect of refining the over-approximations. In the initial formulation of the approach, it returns to the user a set of design locations, where a change can be made to make the target state reachable within a specific number of clock cycles without reaching any other unreachable states. It is additionally extended to handle cases, where the target state is to be reached only after a user-specified number of other unreachable states.

The exact approach eliminates the requirement for the user to specify parameters limiting the types of solutions returned, by merging the steps of SAT-based debugging with PDR-based state space over-approximation. As such, it returns the full solution set to the problem. It constructs an enhanced model of the circuit in which the reachability of particular states implies that specific design locations are solutions. An incremental PDR solver is then used to check the reachability of these states. As a result, the computation of approximations and the detection of spurious results occurs entirely within the PDR solver. Incrementality is applied in a novel fashion that allows nearly all of the solver's internal state to be reused, giving substantial runtime performance gains. Since the approach uses an unbounded model checking technique, it is able to find all solutions to the problem. Additionally, it terminates with a proof of this fact.

The presented techniques are complementary and each provides its own set of strengths and tradeoffs. The approximate approach requires the user to set parameters limiting the portion of the solution space it explores. As a result, it may not find the complete solution set to the problem. It is therefore most applicable in cases, where the user has the requisite information available to select the parameters appropriately, such as when the target state is supposed to be reached in a known number of clock cycles. Under these conditions, it is likely to return the true root cause of the error to the user while achieving better runtime than would be possible with the exact approach. Conversely, the exact approach finds the complete solution set to the problem and does not require the user to select parameters. Ultimately, this provides greater automation and confidence in the results.

Both techniques apply to situations in which a state is unreachable in violation of the design specification, and therefore is indicative of an error in the circuit. In practice, a state may be unreachable for other reasons, such as over-constraining the initial states, or because of intentional design decisions. Our approaches take as input the initial states, the circuit, and the target state. As such, it is assumed these inputs are correct, i.e., the user has correctly identified that the given target state should be reachable from the given initial state according to the design specification. In this case, the only way to correct the error is to modify the circuit, and the presented approaches help the user identify, where to begin making such a change. As is the case for existing SAT-based debugging techniques [10], once a set of solutions is found, the engineer must decide how to implement a fix in order to maintain the desired functionality.

Experiments on `OpenCores` [21] and HWMCC'15 designs are presented demonstrating these tradeoffs and the effectiveness of the presented approaches. The complete solution set to the problem is found to include a median of 1.6% of the design locations, substantially narrowing down the root cause of the error and demonstrating the usefulness of the proposed techniques. The approximate approach offers a substantial 31x speedup compared to the exact approach, while finding 63% of the complete solution set. The incremental application of the model checker is found to provide an impressive 23.9x speedup over the exact approach. Additional experimental results confirm the benefits of the presented performance optimizations.

The remaining sections are organized as follows. Section II presents background information regarding model checking and SAT-based debugging. Section III defines the problem and presents the approximate approach. Section IV presents the exact approach. Section V presents experimental results. Finally, Section VI concludes this paper.

II. Preliminaries

A. Notation

The following notation is used throughout this paper. Given a sequential circuit C, $S = \{s_1, \ldots, s_{|S|}\}$ denotes the set of state elements (registers) of C, while $S' = \{s'_1, \ldots, s'_{|S|}\}$ denotes the set of next-state variables (inputs to registers). For a propositional formula P over the set of state variables S, the primed formula P' represents the same formula over the next-state variables S'. Each assignment $t \in \{0, 1\}^S$ to the state elements represents a state of C. A state can be represented by a cube over the state elements.

The transition relation of C is denoted $T \subseteq \{0, 1\}^S \times \{0, 1\}^S$. For a state pair $\langle t_1, t_2\rangle$, $\langle t_1, t_2\rangle \in T$ if and only if there exists an assignment to the primary input of C that causes a state transition from t_1 to t_2. We assume T is given in a form such that the propositional formula $t_1 \wedge T \wedge t'_2$ is satisfiable if and only if $\langle t_1, t_2\rangle \in T$. This formula constrains T with t_1 at its current-state variables (as t_1 is not primed), and t_2 at its next-state variables (as t_2 is primed). The set of initial states of C is denoted $I \subseteq \{0, 1\}^S$. It is represented by a propositional formula over the state variables also called I, as these are merely different representations of the same thing. If P is a formula over the state variables, a state t is said to be a P-state if and only if t satisfies P.

For the purpose of model checking, C can be modeled by a finite state machine (FSM) $M = (S, I, T)$, where S represents the state elements, I is a propositional formula representing the initial states, and T is the transition relation. A sequence of states $t_0, \ldots, t_n$ is a *trace* of M if and only if $\langle t_i, t_{i+1}\rangle \in T$ for all $0 \leq i < n$ and t_0 is an I-state. A state t is *i-step reachable* if it appears in a trace of i cycles or less from an initial state. We say that a state is reachable if there is a value of i for which it is i-step reachable.

B. SAT-Based Debugging

When verification reveals a safety property failure, the resulting error-trace can be used with an SAT-based automated

debugging tool [10] to aid the engineer in finding the source of the error. The work of [10] operates as follows. Given an error-trace and an error cardinality $n \geq 1$, it returns n-tuples of locations, where a change can be implemented to correct the erroneous behavior exposed by the trace. Letting $L = \{l_1, l_2, \ldots, l_{|L|}\}$ denote the suspect locations in the circuit, the transition relation is enhanced by the addition of a set of *error-select lines* $E = \{e_1, e_2, \ldots, e_{|L|}\}$. When $e_i = 0$ the behavior of the circuit at location l_i is unchanged. When $e_i = 1$, l_i is replaced by a free variable w_i.

For an error-trace of k clock cycles, the enhanced transition relation is then unrolled into an iterative logic array (ILA) representation [22] with k time-frames. Additional constraints are added in each time-frame to set the primary input to the values from the error-trace and to force the primary output to the known reference (correct) values according to the specification. An additional constraint is added to the state variables in the first time-frame to force the circuit to start in a particular initial state. Finally, a cardinality constraint ϕ_n is added to ensure that exactly n error-select lines are simultaneously active. The ILA and constraints are converted into a propositional formula in conjunctive normal form (CNF). This construction is such that each satisfying assignment to the formula indicates an n-tuple of suspect locations, where a change could correct the erroneous behavior demonstrated by the error-trace. An all-solutions SAT solver is used to find all satisfying assignments to the formula.

Considering both the work of [10] and the techniques presented later in this paper, solutions indicate locations, where replacing the Boolean function with a different one can correct the observed failure. The function may be arbitrarily complex, it may depend on arbitrary input, and it may require adding new registers to the circuit. It does not, however, require modifying any other locations in the circuit. Other techniques, such as those based on binary decision diagrams [23] or partial equivalence checking [24] could be used to compute the actual function.

C. Property Directed Reachability

This paper makes extensive use of the unbounded model checking technique of PDR [2], [7]. Given an FSM $M = (S, I, T)$ and a safety property P in CNF that represents the set of safe states, PDR attempts to prove that P is invariant for M (i.e., every reachable state is a P-state). If P is not invariant, a counter-example trace reaching a $\neg P$-state is returned. Alternatively, P is invariant and PDR returns an *inductive invariant* proving this fact.

At a high level, PDR operates as follows. It maintains a sequence of formulas over the state elements $F = \langle F_0, F_1, \ldots, F_k \rangle$, referred to as the *inductive trace*. Each F_i is in CNF, and F_0 is simply set to the initial states I. For all $i > 0$, each F_i over-approximates the post-image of F_{i-1} ($F_{i-1} \wedge T \Rightarrow F_i'$). Since F_0 includes all initial states, this means that F_i over-approximates the i-step reachable states. The fact that F_i over-approximates the i-step reachable states also implies that each clause of F_i over-approximates the i-step reachable states. As such, F_i and each of its clauses are referred to as *i-step invariants*. Additionally, each clause c of each F_i includes every initial state ($I \Rightarrow c$). That is, c satisfies *initiation*.

The algorithm conducts a series of iterations $k = 1, 2, \ldots$ in which iteration k seeks a $(k+1)$-step counter-example. Iteration k consists of solving a series of SAT instances of the form $F_k \wedge T \wedge \neg P'$. If this formula is satisfiable, F_k contains a state t that is one step from violating the property. This does not imply the existence of a counter-example, as it is not known whether t is k-step reachable or merely a spurious artifact of the over-approximation. Therefore, when such a state is found, this essentially triggers a recursive call to PDR intended to determine if t is k-step reachable. If it is indeed k-step reachable, a counter-example is discovered and the algorithm returns REACHABLE. If not, a clause is conjoined to F_k that excludes t from being an F_k-state. The recursive calls may result in additional clauses being added to some or all of the formulas $F_1, \ldots, F_{k-1}$.

Iteration k concludes when F_k contains no predecessors of unsafe states. This condition is detected when $F_k \wedge T \wedge \neg P'$ is unsatisfiable. At this point, the algorithm attempts to construct a proof that P holds from the inductive trace. If a proof is not found, it continues to the next iteration. Throughout this paper, it is assumed that an algorithm PDR(M, P, k) exists, where $M = (S, I, T)$ is an FSM. It returns REACHABLE if and only if a P-state is k-step reachable under M. If $k = \infty$, it returns REACHABLE if and only if a P-state is reachable under M. Otherwise, it returns UNREACHABLE.

III. Approximate Unreachability Diagnosis

This section presents an approximate approach to localize errors that manifest in the form of erroneously unreachable states. The algorithm takes as input a set of suspect locations $L = \{L_1, L_2, \ldots, L_{|L|}\}$, an unreachable target state condition $\mathcal{S}$ represented by a CNF formula, and error cardinality n. The target state condition $\mathcal{S}$ is a formula in CNF such that all $\mathcal{S}$-states are unreachable, and is provided by the user. The set of suspect locations L is similarly provided by the user. In the worst case it can include every location in the circuit. As a practical consideration, a larger suspect set is expected to increase runtime. Therefore, it may be beneficial for the engineer to apply knowledge regarding the source of the error to restrict L to, e.g., all locations within a particular block that is suspected to be the root cause of the error.

A *solution* of error cardinality n is defined as an n-tuple of suspect locations that can be replaced by different Boolean function(s) to make some $\mathcal{S}$-state(s) reachable. The required functions may be arbitrarily complex, but do not require modifying any other locations in the circuit. The purpose of the algorithm is to determine which of the suspect locations are indeed solutions to the problem. As such, the algorithm returns a solution set $L_{\text{sol}} \subseteq L^n$. Note that the algorithm merely indicates locations, where a correction can be made. It is the responsibility of the engineer to decide how to implement the desired change.

In its initial formulation, the algorithm uses steps of state space over-approximation and debugging to return solutions

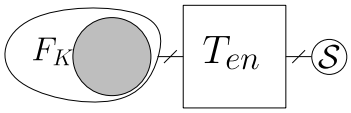

Fig. 1. Model used in SAT-based debugging.

that can make the target state reachable one transition after some already-reachable state. It is later extended to find solutions that make it reachable only after a user-specified number of transitions following a reachable state. A final extension is presented that removes the dependence on fixed approximations to find a larger portion of the solution set.

A. Single-Cycle Approximate Unreachability

This section presents the initial formulation of the methodology. At a high-level, the algorithm models and debugs a single state transition from an already-reachable state to a target state. As computing the exact set of reachable states is an intractable problem, the over-approximations computed by PDR are used instead. Due to the inherent nature of the approximations, it is possible to find spurious solutions, necessitating an extra step to detect them. The formulation presented in this section also requires the user to provide as input a parameter K known as the *cycle limit*. It seeks solutions that can be used to make a target state $(K+1)$-step reachable.

In greater detail, the algorithm involves three main steps: 1) reachable state space over-approximation; 2) debugging; and 3) spurious solution detection. The reachable state space over-approximation step computes the initial approximation of the set of K-step reachable states. This is done by executing PDR directed toward the goal of proving that $\mathcal{S}$ is not K-step reachable, which produces the needed approximation in the form of the formula F_K from the inductive trace.

As the next step, the algorithm constructs an SAT-based debugging instance toward the goal of finding an n-tuple of suspect locations that can be changed to allow for a transition from some F_K-state to some $\mathcal{S}$-state. As such, this step constructs a propositional formula in the manner described in Section II-B. Letting T_{en} represent the enhanced transition relation with added error-select lines for each suspect location in L, the resulting propositional formula is

$$F_K \wedge T_{\text{en}} \wedge \mathcal{S}' \wedge \phi_n. \tag{1}$$

Intuitively, the debugging instance of (1) consists of a single copy of the transition relation, constrained with F_K as its current state and $\mathcal{S}$ as its next state. The primary input and output are left unconstrained, allowing the SAT solver to find solutions for any input assignment. A cardinality constraint ϕ_n is used to find solutions of cardinality n. This model is depicted graphically in Fig. 1, where the shaded region represents K-step reachable states.

A satisfying assignment to (1) represents an n-tuple of locations that can be changed to make an F_K-state transition to an $\mathcal{S}$-state. Due to the inherent nature of F_K, this may not be a solution, as the chosen F_K-state may not actually be reachable. This means that some satisfying assignments of (1) may not correspond to solutions as defined earlier. These are referred to as *spurious solutions*.

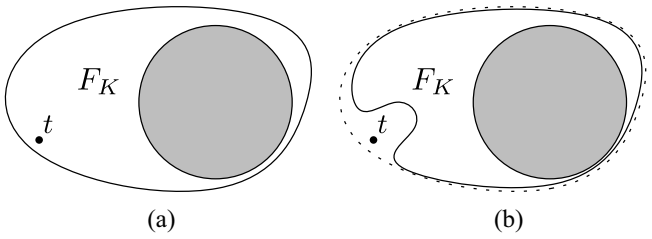

Fig. 2. F_K-states (a) initially and (b) after spurious result from state t.

This kind of result is detected and rejected in the spurious solution detection step as follows. Let t represent the chosen F_K-state. When a satisfying assignment is found to the formula, it is verified by using PDR to determine if state t is K-step reachable. If so, the solution is proven to be nonspurious, and is therefore added to the solution set. Additionally, a blocking clause is added to the debugging instance to ensure that the same solution is not found again. If t is found not to be K-step reachable, the solution is discarded. As a side effect, PDR refines its approximations and F_K no longer includes state t.

In practice, when a spurious solution is detected, the generalization done by PDR may remove many other states that are not K-step reachable from F_K. This is shown in Fig. 2, where after detecting a spurious result from state t, the more accurate approximation shown in Fig. 2(b) is derived. This tends to result in a rapid increase in the accuracy of the approximations. The increased accuracy is expected to reduce the chances of finding further spurious solutions.

Note that the spurious solution detection step may discard nonspurious solutions. A solution that was found using current state t may be discarded despite being nonspurious in two cases. First, it may be the case that a fix can *both* make t reachable and make an $\mathcal{S}$-state reachable from t. The extension presented in the next section uses a debugging instance with multiple time-frames to find solutions that require reaching other unreachable states before the chosen target state, and is therefore better able to handle this case.

Next, it may be the case that state t is not K-step reachable but is reachable in a larger number of cycles. This could be remedied by using an unbounded call to PDR rather than a check for K-step reachability in the spurious solution detection step. However, this sacrifices repeatability, as there are random factors used in the generalization procedure of PDR. As a result, different runs of the algorithm may give different approximations. As such, the presence of a state that is not K-step reachable in F_K is dependent on randomness. Therefore, if an unbounded call was used to detect spurious solutions the results may be dependent on the random seed, which is not desirable. The extension presented in Section III-C eliminates this problem, by using an over-approximation of all reachable states in place of F_K.

Pseudocode for the procedure is shown in Algorithm 1. The algorithm is called SCUNREACHABILITY, shorthand for single-cycle unreachability, as it finds solutions that reach the target one cycle after a reachable state. In that description, procedure EXTRACTSTATE extracts the chosen state of F_K from a

Algorithm 1 SCUNREACHABILITY($C, L, \mathcal{S}, K, n$)

1: $L_{sol} = \emptyset$
2: $S_C =$ state element set of C
3: PDR($(S_C, I, T), \mathcal{S}, K$)
4: $U = F_K \wedge T_{en} \wedge \mathcal{S}' \wedge \phi_n$
5: **while** $SAT(U) \neq UNSAT$ **do**
6: $t =$EXTRACTSTATE(*Solution*)
7: **if** PDR($(S_C, I, T), t, K$) = REACHABLE **then**
8: *Solution* = locations with active ES lines from SAT
9: $L_{sol} = L_{sol} \cup \{Solution\}$
10: $B =$ Blocking clause for *Solution*
11: $T_{en} = T_{en} \wedge B$
12: **else**
13: $F_K =$ updated formula extracted from PDR on line 7
14: $U = F_K \wedge T_{en} \wedge \mathcal{S}' \wedge \phi_n$
15: **end if**
16: **end while**
17: **return** L_{sol}

satisfying assignment. Line 3 executes PDR directed at proving $\mathcal{S}$ is unreachable, implicitly computing F_K. Note that we assume $\mathcal{S}$ is unreachable, but additional steps could be taken to prove it, terminating if it is found to be reachable. Line 4 of the algorithm constructs the initial debugging instance for the current iteration. The loop on lines 5–16 repeatedly finds satisfying assignments. Line 7 checks if the found solution could be spurious. If it is a real solution, line 9 records it, while line 11 blocks the solution from being found again by adding a clause to T_{en}. If the solution contains active error-select lines $e_{i_1}, \ldots, e_{i_n}$, the blocking clause is $(\neg e_{i_1} \vee \cdots \vee \neg e_{i_n})$. Otherwise, the solution is discarded and PDR updates F_K to block state t. In this case, line 14 uses the newly updated F_K to update the debugging instance. The iteration continues until the debugging instance is unsatisfiable, at which point line 17 returns the solution set.

This algorithm returns a particularly useful subset of the complete solution set to the problem. Specifically, it returns all solutions, where a change can make a target state reachable one cycle after a K-step reachable state. As demonstrated empirically in Section V, this is often adequate to find the true source of the error. Note that in Algorithm 1, F_K is computed based on the original transition relation T with no error-select lines. If the user implements a change at a solution location, it may change the behavior of the circuit such that the chosen F_K-state is not reachable. As is the case for traditional SAT-based debugging techniques [10], care must be taken to implement the fix correctly. A full verification step may be needed after correcting the design to ensure correctness after fixing the design.

B. Multicycle Approximate Unreachability

The approach of the previous section finds a useful subset of the complete solution set. However, it is unable to find solutions that require more than one transition to reach the target state from some already-reachable state. In order to address this limitation, this section extends the methodology to find solutions, where a user-specified number of transitions are required. As such, it requires an additional input parameter N, referred to as the *window size*. As was the case for the algorithm of the previous section, it requires the cycle limit parameter K.

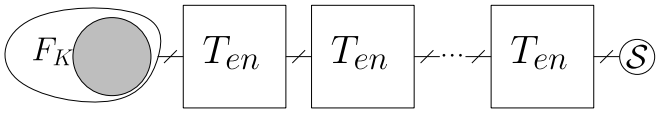

Fig. 3. Model used in SAT-based debugging for the multicycle case.

To achieve the above, this approach uses the same three steps of reachable state space over-approximation, debugging, and spurious solution detection. The key difference is in the debugging step. Rather than using the debugging instance of (1), it models and debugs a sequence of N state transitions that starts at F_K and ultimately transitions to $\mathcal{S}$. As such, it uses an ILA representation of the enhanced transition relation consisting of N time-frames. The current-state of the first time-frame is constrained to F_K, while the next-state of the final time-frame is constrained to $\mathcal{S}$. As before, the primary input and output are left unconstrained. The resulting debugging instance is expressed as follows:

$$F_K \wedge T_{\text{en}}^1 \wedge \cdots \wedge T_{\text{en}}^N \wedge \mathcal{S}' \wedge \phi_n \tag{2}$$

where T_{en}^i denotes the enhanced transition relation in the ith time-frame of the ILA. Intuitively, the cycle limit parameter K represents the number of clock cycles for which the circuit's behavior is modeled by the approximation derived from PDR. The window size N is the number of cycles for which the algorithm is allowed to "look forward" for unreachable states so that it ultimately reaches $\mathcal{S}$. The debugging instance used here is depicted in Fig. 3.

Rather than finding solutions that make a target state $(K+1)$-step reachable, this approach finds solutions that may make the target state $(K+N)$-step reachable. More specifically, it finds the set of all solutions that may make the target state reachable in N or fewer steps after a K-step reachable state. Essentially, this represents a more general version of the approach in the previous section. As demonstrated by empirical results in Section V, this is particularly valuable in the case of pipelined designs.

C. Unlimited Cycle Approximate Unreachability

The previous two sections present approaches, where a fixed cycle limit K is used in modeling the set of potentially reachable states. As mentioned in Section III-A, the algorithm may discard a solution as being spurious when the chosen state from the approximation is only reachable in a number of cycles greater than K. This section extends the methodology to find solutions that would be discarded in that case. The extension presented here finds solutions that make the target state reachable N or fewer steps following any reachable state, and eliminates the need for the cycle limit parameter.

To accomplish this, the reachable state space approximation step of the previous sections is essentially eliminated,

Algorithm 2 UCUNREACHABILITY($C, L, \mathcal{S}, n$)

1: $L_{sol} = \emptyset$
2: $S_C =$ state element set of C
3: $F_\infty = \neg\mathcal{S}$
4: $U = F_\infty \wedge T_{en} \wedge \mathcal{S}' \wedge \phi_n$
5: **while** $(Solution = SAT(U)) \neq UNSAT$ **do**
6: $t =$EXTRACTSTATE($Solution$)
7: **if** PDR($(S_C, I, T), t, \infty$) = REACHABLE **then**
8: $L_{sol} = L_{sol} \cup \{Solution\}$
9: $B =$ Blocking clause for $Solution$
10: $T_{en} = T_{en} \wedge B$
11: **else**
12: $V =$ inductive invariant extracted from PDR
13: $F_\infty = F_\infty \wedge V$
14: $U = F_\infty \wedge T_{en} \wedge \mathcal{S}' \wedge \phi_n$
15: **end if**
16: **end while**
17: **return** L_{sol}

and in place of F_K, an over-approximation of all reachable states is used. This approximation is denoted by F_∞ to distinguish it from the bounded approximations used in the previous sections. The algorithm uses $F_\infty = \neg\mathcal{S}$ as the initial approximation of the reachable state space rather than computing an initial approximation with PDR. Subsequently, the debugging step begins in an attempt to find solutions that allow reaching a target state up to N steps after any F_∞-state. The debugging instance can be expressed as

$$F_\infty \wedge T_{\text{en}}^1 \wedge \cdots \wedge T_{\text{en}}^N \wedge \mathcal{S}' \wedge \phi_n. \tag{3}$$

The primary difference to the approaches of the previous sections is in the spurious solution detection step. Upon finding a satisfying assignment to (3), it is necessary to check if the chosen F_∞-state is reachable using an unbounded call to PDR. As was the case before, if the chosen state is reachable, the solution is recorded and a blocking clause is added to the debugging instance. Conversely, if the chosen F_∞-state is unreachable, PDR returns an inductive invariant proving this fact. As explained in Section II-C, the inductive invariant is a formula in CNF that over-approximates the reachable states. It is conjoined to F_∞ refining the approximation in a manner that excludes the chosen unreachable state and possibly excludes many other unreachable states.

This approach represents a half-step between the approximate approaches and the exact approach of the next section. While this approach considers F_∞, effectively similar to setting K to infinity, it still returns a subset of the complete solution set. Finding the complete solution set requires considering values of N up to infinity, or equivalently, considering the possibility that an arbitrary number of currently unreachable states may need to be reached prior to reaching $\mathcal{S}$. The exact approach of the next section accomplishes this goal by executing PDR against T_{en} rather than T.

Pseudocode for the procedure is shown in Algorithm 2. It is presented as an extension of Algorithm 1, but can easily be modified to support a window size parameter as well. The primary difference from Algorithm 1 is in lines 12 through 14. On line 12, the inductive invariant computed by PDR is extracted. Subsequently, line 13 refines the over-approximation of all reachable states using the inductive invariant. Additionally note the absence of the initial call to PDR before executing the debugging step. Otherwise, the algorithm is structured similarly to Algorithm 1.

IV. EXACT UNREACHABILITY DIAGNOSIS

The approximate approach presented in the previous section finds a particularly useful subset of the complete solution set. However, it has two drawbacks. The first is that in order to use it, the engineer must apply design knowledge to intelligently select parameter values in order to balance runtime with completeness of the solution set. The second is that it does not guarantee that the returned solution set includes the actual error source, limiting confidence in its results. While the unbounded version presented in Section III-C somewhat mitigates these issues by eliminating the cycle limit parameter, it still is fundamentally limited due to the dependence on the window size parameter. This section presents a methodology that removes these drawbacks for one who is willing to trade resolution for run-time performance.

The work of [25] presents a parameter synthesis technique for infinite-state transition systems using generalized PDR with a satisfiability module theories solver [26]. The technique determines parameters that guarantee a system maintains desired safety properties. The parameters are analogous to initial states in our approach. It operates by starting with the parameters unconstrained and repeatedly finding counter-examples. The counter-examples are used to refine the parameter values, and the process continues until the parameters are such that the desired safety properties hold. By carefully constructing the parameters (i.e., initial states) and transition relation, the technique proposed in this section represents a specialized application of the technique in [25].

The proposed methodology solves a series of unbounded model checking instances using PDR. The instances are constructed using an enhanced FSM model of the circuit with added hardware to facilitate diagnosis. An additional feature key to the runtime performance of our approach is the incremental use of the model checking algorithm. For ease of presentation, the model checker is treated as a "black box" until Section IV-C, which discusses internal behavior of the model checker to clarify and justify the use of incrementality. Section IV-D presents further optimizations dependent on the internal behavior of the model checker.

A. Enhanced Model Construction

The enhanced FSM model behaves like the original circuit with specific suspect locations replaced by unknown Boolean functions. More specifically, the number of suspect locations replaced is equal to the error cardinality n. Which suspect locations are replaced depends on value assignments to the *error-select registers*, which are new hardware added to the circuit to facilitate diagnosis. They are similar to the error-select lines used in SAT-based debugging and serve a similar

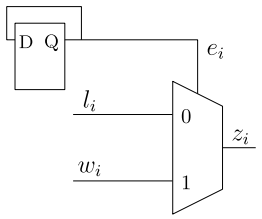

Fig. 4. Error select register and multiplexer at suspect location l_i.

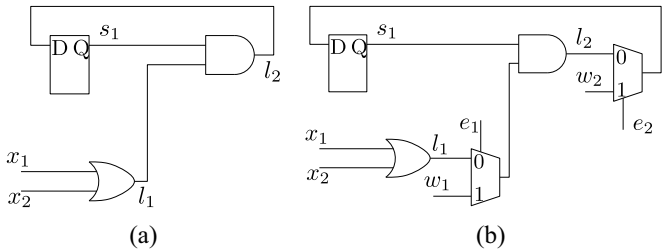

Fig. 5. (a) Original circuit. (b) Circuit of T_{en} (added registers omitted).

purpose, but are suitable for use in an unbounded model checking problem. In particular, error-select lines are more similar to inputs than to registers. However, using inputs to control the replacement of suspect locations would allow the PDR solver to choose different locations in each clock cycle. Using registers and other hardware explained later in this section, the problem instances are crafted such that the replacement is consistent across clock cycles.

The enhanced model is denoted $M = (S \cup E, I_{\text{en}}, T_{\text{en}})$, and is constructed from that of the original circuit by adding a set of error-select registers $E = \{e_1, \ldots, e_{|L|}\}$ and constructing enhanced transition relation T_{en} along with the enhanced initial state condition I_{en}. Given a trace of the enhanced model $t_{M,0}, \ldots, t_{M,m}$, the original circuit is said to have an equivalent trace $t_{C,0}, \ldots, t_{C,m}$ if and only if the original registers in the set S have the same value assignments in state $t_{M,i}$ and $t_{C,i}$ for all $0 \leq i \leq m$.

The enhanced transition relation is constructed from that of the original circuit by inserting additional hardware to facilitate diagnosis. For each suspect location l_i, an associated error-select register e_i, and free variable w_i are added. Subsequently, new hardware is constructed such that if $e_i = 1$, l_i is effectively disconnected from its fanout and replaced by w_i. If $e_i = 0$, the circuit's behavior is unaffected. In other words, the value assigned to e_i controls whether or not l_i is replaced by a free variable. As is explained later, an important aspect of the enhanced model's behavior is that the chosen initial state from I_{en} dictates which suspect locations are replaced by free variables. As a result, while the model checker is free to choose any initial state from I_{en}, the values for each e_i are not allowed to change during state transitions. This necessitates a constraint enforcing that $e_i' = e_i$ for all error-select registers. Without this constraint, the values assigned to the error-select registers would be allowed to change during state transitions, and therefore the initial state would not dictate which suspect locations are replaced.

This construction can be implemented using a multiplexer and a register. The multiplexer has 0-input l_i, 1-input w_i, and select line e_i. The output of the multiplexer is denoted z_i and is connected to the original fanout of l_i. This enforces the required behavior, where the value assigned to e_i controls whether or not l_i is replaced by a free variable. The register enforces the constraint that $e_i' = e_i$. This can be implemented by feeding its output back to its input. Fig. 4 depicts this multiplexer and error-select register construction. The enhanced transition relation is derived from the circuit with the added hardware. The following example illustrates the behavior of T_{en} and will be used to explain various aspects of the algorithm throughout this section.

Example 1: Consider the circuit of Fig. 5(a). It has a single state element s_1, two primary input signals x_1 and x_2, and two suspect locations are indicated as l_1 and l_2. Assume that the initial state is represented by the cube $\bar{s_1}$ [i.e., $I = (\bar{s_1})$]. It can easily be verified that it is impossible to reach a state in which $s_1 = 1$. To diagnose this unreachability, given $\mathcal{S} = (s_1)$, $L = \{l_1, l_2\}$, and $I = (\bar{s_1})$, the enhanced transition relation is constructed from the circuit shown in Fig. 5(b). When $e_1 = e_2 = 0$, this circuit behaves identically to the original. When $e_1 = 1$, l_1 is replaced by the free variable w_1, which allows it to assume any value during model checking. This effectively replaces l_1 with an arbitrary unknown Boolean function. Similar behavior applies to l_2 and e_2.

The only remaining component of the enhanced model is I_{en}. As mentioned earlier, we associate the reachability of particular states under the enhanced model with a specific n-tuple of locations being a solution. Toward the goal of constructing I_{en}, consider a trace of the enhanced model. As T_{en} has a constraint enforcing $e_i' = e_i$ for all error-select registers, all states in the trace must have the same active error-select registers $e_{i_1}, \ldots, e_{i_m}$. The enhanced model therefore behaves like the original circuit with $l_{i_1}, \ldots, l_{i_m}$ replaced by free variables. It can be concluded that the original circuit has an equivalent trace if the Boolean functions at those locations are simultaneously replaced by different functions. If this trace contains an $\mathcal{S}$-state, then simultaneously replacing $l_{i_1}, \ldots, l_{i_m}$ makes a target state reachable.

Now consider a trace of the enhanced model that starts from an I-state, ends on an $\mathcal{S}$-state, and has exactly n active error-select registers $e_{i_1}, \ldots, e_{i_n}$. Using the argument from the previous paragraph the original circuit has an equivalent trace when $l_{i_1}, \ldots, l_{i_n}$ are replaced with unknown Boolean functions. The equivalent trace starts from an initial state and ends at a target state, so replacing these n locations makes a target state reachable. In other words, $l_{i_1}, \ldots, l_{i_n}$ is a solution of cardinality n. This argument applies to any trace satisfying these three properties. The algorithm is intended to find such traces.

This motivates the construction of the enhanced model's initial state formula I_{en}. The original registers of the circuit are constrained with the original initial state formula I. This ensures that traces of the enhanced model begin on an I-state. Since exactly n error-select registers must be active, the error-select registers are constrained using a cardinality constraint ϕ_n. The enhanced initial state formula is therefore $I_{\text{en}} = I \wedge \phi_n$.

This completes the construction of the enhanced model, though it still remains to tailor the algorithm to find traces with the final requirement of ending on an $\mathcal{S}$-state. Intuitively, this is handled using the property given to the model checker, as explained later. The following example builds on the previous one to clarify the behavior of the enhanced model.

Example 2: Consider once again the circuit of Fig. 5. Assuming an error cardinality of one, the enhanced initial state condition is $I_{\text{en}} = I \wedge \phi_1$. Therefore, $I_{\text{en}} = (\bar{s_1}) \wedge (e_1 \vee e_2) \wedge (\bar{e_1} \vee \bar{e_2})$. Representing states as cubes, the set of states satisfying I_{en} is $\{(\bar{s_1} \wedge e_1 \wedge \bar{e_2}), (\bar{s_1} \wedge \bar{e_1} \wedge e_2)\}$. Notice that these are all states in which $s_1 = 0$, corresponding to initial states of the original circuit. Additionally, every I_{en}-state has exactly one active error-select register. Therefore, these states meet the requirements for initial states of traces that indicate solutions.

Algorithm 3 UNREACHABILITY($C, L, \mathcal{S}, n$)

1: $L_{sol} = \emptyset$
2: $S_C =$ state element set of C
3: $T_{en}, E =$ CONSTRUCTMODEL(L, C)
4: $I_{en} = I \wedge \phi_n$
5: $M = (S_C \cup E, I_{en}, T_{en})$
6: **while** $PDR(M, \mathcal{S}, \infty) ==$ REACHABLE **do**
7: $\quad e_{i_1}, ..., e_{i_n} =$ active error-select registers in trace
8: $\quad L_{sol} = L_{sol} \cup \{(l_{i_1}, ..., l_{i_n})\}$
9: $\quad B = (\neg e_{i_1} \vee ... \vee \neg e_{i_n})$
10: $\quad M_{blk} = (S_C \cup E, I_{en} \wedge B, T_{en})$
11: $\quad M = M_{blk}$
12: **end while**
13: *invariant* = inductive invariant extracted from PDR
14: **return** (L_{sol}, *invariant*)

B. Finding Solutions With PDR

As mentioned earlier, the final requirement for a trace to indicate a solution is that it must end on an $\mathcal{S}$-state. This is accomplished simply by using $\mathcal{S}$ as the unsafe state formula when calling PDR. If any target state is reachable, then PDR will return REACHABLE along with a counter-example trace that meets the requirements previously described. If $e_{i_1}, \ldots, e_{i_n}$ are the active error-select registers in the counter-example, then $l_{i_1}, \ldots, l_{i_n}$ is a solution. The following example demonstrates the process of finding a solution.

Example 3: Continuing the illustration of the methodology from the previous example, recall that the target state condition is $\mathcal{S} = (s_1)$ and the initial state condition is $I = (\bar{s_1})$. The enhanced model has the following counter-example trace: $\langle t_0, t_1 \rangle = \langle (\bar{s_1} \wedge \bar{e_1} \wedge e_2), (s_1 \wedge \bar{e_1} \wedge e_2) \rangle$. Notice that t_0 corresponds to an initial state of the original circuit, t_1 is a target state, and e_2 is the active error-select register. In states t_0 and t_1 the model behaves identically to the original circuit with l_2 replaced by an unknown function. Since t_0 is an initial state and t_1 is a target state, replacing l_2 with a different function makes a target state reachable in the original circuit. This indicates that location l_2 is a solution. Indeed, the reader can confirm that replacing the AND-gate that drives l_2 with an OR-gate makes the target state reachable. Other corrections to the problem are also possible.

After a finding a solution, it is necessary to continue searching for additional solutions, if any. This is accomplished by modifying I_{en} to exclude a solution after it is found. If a solution $l_{i_1}, \ldots, l_{i_n}$ is found, then I_{en} is updated by conjoining the blocking clause $(\neg e_{i_1} \vee \cdots \vee \neg e_{i_n})$. This prevents PDR from finding any further counter-examples in which all of those same error-select registers are active. By repeating this procedure, eventually the algorithm will reach a point, where no I_{en}-state can reach an $\mathcal{S}$-state. When this occurs, the algorithm terminates. The following example builds on the previous one to demonstrate the procedure of blocking a solution and terminating.

Example 4: In the previous example, the solution l_2 was found. After blocking it by conjoining the clause $(\neg e_2)$, the enhanced initial state condition becomes $I_{\text{en}} = (\bar{s_1}) \wedge (e_1 \vee e_2) \wedge (\bar{e_1} \vee \bar{e_2}) \wedge (\bar{e_2})$, leaving $(\bar{s_1} \wedge e_1 \wedge \bar{e_2})$ as the only remaining initial state. It is easily verified that this state cannot reach any target states. Therefore, we can infer that l_1 is not a solution. This is indeed the case. To reach a state, where $s_1 = 1$ the output of the AND-gate must be 1. In the initial state $s_1 = 0$, so regardless of the value at l_1 the AND-gate will never output 1. Therefore, there is no way to modify the circuit at l_1 to rectify the unreachability of the target state.

Pseudocode for the procedure is shown in Algorithm 3. In that description, algorithm CONSTRUCTMODEL receives input L and C and returns the enhanced transition relation and error-select register set. Lines 3–5 construct the enhanced FSM model that is used by PDR. Lines 6–12 contain the main loop in which solutions are found. If a solution exists, it is extracted (line 7) and added to L_{sol} (line 8). Line 10 constructs a new model M_{blk} in which the solution is blocked, while the next line updates M. The distinction between M and M_{blk} is intended to simplify the discussion in Section IV-C. As the number of suspect locations is finite, the loop will eventually terminate. At this point, PDR indicates $\mathcal{S}$ is unreachable and an inductive invariant is extracted (line 13). Finally, L_{sol} and the proof of solution completeness are returned in line 14.

C. Incremental PDR

The previous section treats PDR as a black box to maintain conceptual simplicity. However, we note that the model changes only very slightly between consecutive calls to PDR. For a small change in the model, it is expected that many of the invariants may remain valid [27] and therefore substantial performance gains may be achieved by applying PDR incrementally. This section uses the internal behavior of PDR and the structure of Algorithm 3 in order to explain the incremental use of the model checking algorithm. In this context, incrementality means that each call to the model checker reuses the inductive trace from the previous run. This is of critical importance to the runtime of the algorithm, as in Algorithm 3, each solution requires an additional call to PDR. In the worst case, this requires a total of $\binom{|L|}{n}$ calls, suggesting that the performance gains that can be achieved with incrementality may be significant.

As explained in Section II-C, PDR maintains a sequence of CNF formulas $F = \langle F_0, \ldots, F_k \rangle$ called the inductive trace. Each F_i and each clause c of F_i are i-step invariants. Additionally, they satisfy initiation, meaning that $I_{\text{en}} \Rightarrow c$. The work of [27] presents an invariant finder that determines which invariants computed under one model are also invariant under another model. This provides a means for the reuse of a portion of the inductive trace after changing the model by, e.g., modifying the initial states as Algorithm 3 does. It involves executing a series of SAT queries to determine which clauses are usable with the new model. However, it is in fact possible to exploit the structure of the model updates in Algorithm 3 to reuse every clause without the additional verification steps required by [27]. To demonstrate this, we show that after updating I_{en}, for each clause c in each formula F_i, the updated $I_{\text{en}} \Rightarrow c$ and c is i-step invariant for the new model.

Consider the state of Algorithm 3 immediately after executing line 10. The first requirement is that $I_{\text{en}} \wedge B \Rightarrow c$ for every clause c of every formula F_i of the inductive trace. This follows immediately from the fact that $I_{\text{en}} \Rightarrow c$ by the behavior of PDR and that $(I_{\text{en}} \wedge B) \Rightarrow I_{\text{en}}$ trivially.

The proof of the latter requirement (i.e., each clause c of F_i is also i-step invariant for M_{blk}), ultimately arises from the fact that the reachable state set of M_{blk} is a subset of that of M. This implies that any over-approximation of the reachable states of M also over-approximates the reachable states of M_{blk}. An i-step invariant simply over-approximates the i-step reachable states, and so intuitively one would expect the i-step invariants from M to also hold for M_{blk}. Lemma 1 provides a first step toward proving this claim by showing that conjoining the blocking clause B from line 9 of Algorithm 3 to I_{en} does not make any previously unreachable states become reachable.

Lemma 1: All B-states that are not i-step reachable under M are not i-step reachable under M_{blk} for all $i \geq 0$.

Proof: Consider a B-state t that is not i-step reachable under M. Assume toward a contradiction that it is i-step reachable under M_{blk}. For some $m \leq i$ the model M_{blk} must have a trace $t_0, \ldots, t_m$, where t_0 is an $(I_{\text{en}} \wedge B)$-state and $t_m = t$. As all literals of B are error-select registers and t is B-state, t_0 is also a B-state. This is because the error-select registers cannot change their value assignments.

Both models M and M_{blk} have the same transition relation. Therefore, each transition in the trace is valid under M. As a result, t is only unreachable under M if t_0 is not an I_{en}-state. This is a contradiction as t_0 is an $(I_{\text{en}} \wedge B)$-state and it has already been shown that $(I_{\text{en}} \wedge B) \Rightarrow I_{\text{en}}$. Therefore, all B-states that are not i-step reachable under M are not i-step reachable under M_{blk}. ■

As the lemma shows, blocking a solution in Algorithm 3 does not make any unreachable B-states reachable. Further, it clearly makes all $\neg B$-states unreachable. These two facts together imply that no unreachable states of M_{blk} are reachable under M. Allowing R (R_{blk}) to denote the set of reachable states under M (M_{blk}), clearly $R_{\text{blk}} \subseteq R$. It still remains to show how this implies that invariants of M are invariants of M_{blk}. To provide some intuition behind this reasoning, consider a clause c that is invariant for M, which implies that it over-approximates R. It must also over-approximate R_{blk}, as depicted in Fig. 6.

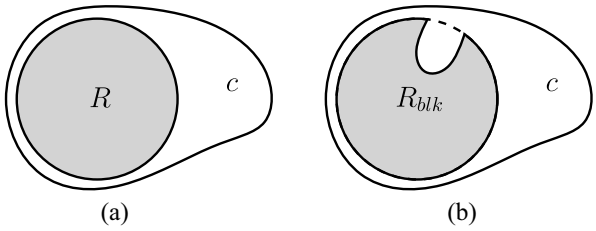

Fig. 6. State space representation of (a) M and (b) M_{blk}.

The discussion above focused on invariants of M, but the same reasoning applies to i-step invariants as demonstrated by the following theorem.

Theorem 1: All clauses that are i-step invariant under M are i-step invariant under M_{blk}.

Proof: Let c be a clause that is i-step invariant under M. Assume toward a contradiction that c is not i-step invariant under M_{blk}. This implies that there is a $\neg c$-state t that is i-step reachable under M_{blk}. Additionally, since c is i-step invariant for M and t is a $\neg c$-state, t must not be i-step reachable under M.

Since t is i-step reachable under M_{blk} and not M, by Lemma 1 it is a $\neg B$-state. No $\neg B$-states are reachable under M_{blk}, contradicting the assumption that c is not i-step invariant under M_{blk}. ■

Theorem 1 proves that it is possible to reuse the entire inductive trace from previous calls to PDR without applying any verification steps. That is, the execution of PDR on line 6 of Algorithm 3 can be done incrementally. This results in a substantial reduction of the algorithm's runtime, as demonstrated by empirical results presented in Section V.

D. Performance Optimization

The algorithms presented in this paper make extensive use of incremental PDR, and as demonstrated by results in Section V, they gain substantial performance benefits from incrementality. However, each clause in the inductive trace incurs some runtime overhead within the PDR solver. In particular, the proof detection step attempts to push clauses forward from F_i to F_{i+1} for $1 \leq i \leq k$ in an effort to produce an inductive invariant. This requires an SAT query for each clause in the inductive trace, which can have a substantial runtime cost. This can be particularly expensive when applying incremental PDR, as each model checking query tends to add new clauses to the inductive trace.

The model checking algorithm of Quip [6] is based on PDR, but adds additional reasoning capabilities. Critically, it has a concept of "bad" clauses, which are clauses that will never appear in an inductive invariant. This occurs when a clause c excludes a reachable state, i.e., a $\neg c$-state is reachable. Intuitively, if a $\neg c$-state is reachable, c cannot appear in a CNF formula that over-approximates the reachable states, meaning c is bad. When this condition is detected, there is no need to attempt to push clause c forward, as it will never appear in the proof.

TABLE I
DETAILS OF BENCHMARK CIRCUITS

benchmark	#gate	$\|L\|$	#sol	%sol
divider	3555	3915	38	0.97%
mrisc_core	8206	9572	18	0.19%
spi	1020	1156	23	1.99%
usb_core	5010	5545	6	0.11%
wb	390	451	193	42.8%
sudoku	66650	67787	2	0.003%
or1200	93430	106534	52	0.05%
shift1add256	93	113	59	52.2%
shift1add512	98	119	59	49.6%
cmugigamax	615	662	299	45.2%
bjrb07amba1	1025	1051	40	3.81%
bobuns2p10	20783	20883	230	1.10%
AVERAGE				16.5%
MEDIAN				1.55%

In Algorithm 3, we therefore use a modified incremental PDR solver that incorporates this feature. When failing to push a clause forward, the solver performs an extra query to detect if the clause is bad. As is the case for [6], this essentially involves a recursive call to PDR. For a clause c that cannot be pushed from F_i to F_{i+1} under model M, the PDR query is PDR($M, \neg c, i+1$). If this query returns REACHABLE, then a $\neg c$-state is $(i+1)$-step reachable and c is detected to be bad. Further attempts to push it forward will be skipped, as it will never be pushed successfully. Conversely, if the query returns UNREACHABLE, then c is successfully pushed forward.

However, as more solutions are found more states become unreachable. In particular, after blocking a solution l_i, all states in which $e_i = 1$ are made unreachable. Some clauses that are marked bad may no longer be bad after blocking. Two approaches are suggested to deal with this. The first is to simply ignore it and allow the solver to relearn such clauses on an as-needed basis. The second is to note which error-select registers are active in a counter-example trace that leads to the detection of a bad clause. Subsequently, when a solution is blocked, any clauses that were marked bad due to a counter-example with the same active error-select registers are unmarked. In the experiments of Section V, the latter approach is used, as the overhead of doing so is minimal.

Additionally, when the above query returns REACHABLE, the counter-example trace returned by PDR contains a set of reachable states. Each clause in the inductive trace is compared against all of the states in the trace. If any clause excludes any known reachable state, it is marked as bad. These states are also stored to be compared against clauses learned in the future. Ultimately, this optimization mitigates the runtime overhead from the constantly growing number of clauses. However, it may also introduce extra overhead from the added bad clause detection queries. For this optimization to be successful, we expect two conditions must be met. The first is that numerous bad clauses are detected. The second is that the inductive trace found by PDR is long, meaning that many attempts are made to push lemmas forward. Under these two conditions, the amount of run-time saved by not trying to push bad lemmas is expected to outweigh the added overhead.

V. EXPERIMENTS

All results presented in this section are executed on a single core of a workstation running Linux with an i5-3570 CPU clocked at 3.4 GHz and 16 GB of RAM. The proposed algorithms are implemented using a state-of-the-art SAT-based debugging algorithm [10] and a reference implementation of PDR [2]. Experiments are timed out after 12 h. Five problem instances from the HWMCC'15 safe track are used, along with seven derived from OpenCores designs [21]. The hardware model checking competition (HWMCC) problem instances are constructed directly from the HWMCC circuits, with the goal of finding solutions to make the given bad (i.e., unreachable) state reachable. The OpenCores problem instances are constructed by manually injecting common design errors that make at least one state unreachable. The chosen design errors are those typically observed in industry, such as complemented conditions in if-statements, changed operators, etc. The suspect set L is chosen to include every design location in the cone-of-influence of the target state. All experiments are executed using error cardinality $n = 1$. Higher error cardinalities can be handled in the same manner as in the work of [10], and as such are not a contribution of this paper.

A bounded model checking (BMC)-like approach was also implemented to compare against the approximate approach. From each value of M from 1 to $(K+N)$, M time-frames are unrolled. If $M \leq N$, all M time-frames are constructed from T_{en}. Otherwise, $(M-N)$ time-frames are constructed from T and N from T_{en}. This mimics the approach of Section III-B and finds the same solution set, but replaces the PDR-derived approximations with concrete instantiations of the transition relation.

When running the exact approach on circuits with 10 000 or more suspects, suspects are checked in batches of 5000. That is, 5000 arbitrary suspects are put in the set L and the algorithm is executed. After it terminates, 5000 different suspects are chosen. This repeats until every suspect has been considered. This is necessary because each suspect location introduces a register into the design, which can cause substantial slowdowns in PDR for large values of $|L|$. Limiting the number of suspects in this manner presents a tradeoff between repeated computation due to aspects of the problem that are common to the different suspect batches, and the slowdowns from adding more registers. This approach still finds the complete solution set to the problem, it merely represents a performance-tuning heuristic. It may be possible to identify approaches to batch the suspects in a manner that further improves performance, but in this paper they are grouped into batches arbitrarily. Grouping the suspects into batches in a manner that improves performance is a topic of future work.

Table I shows the details of the benchmark circuits. The five columns show the name of the problem instance, number of gates, number of suspect locations, number of solutions, and percentage of suspect locations that are solutions, respectively. The number of gates is derived from the circuit's and-inverter graph [28] representation, and gates that are not in the cone-of-influence of the registers defining the target state are removed. Table II shows comprehensive results. The first column shows

TABLE II
RUNTIME AND SOLUTIONS FOUND

	APPROXIMATE												EXACT
	$K = 50, N = 1$			$K = 20, N = 1$			$K = 16, N = 5$			$K = \infty, N = 1$		$K = \infty, N = 5$	
benchmark	spee dup	BMC spee dup	% sol	spee dup	BMC spee dup	% sol	spee dup	BMC spee dup	% sol	spee dup	% sol	spee dup	% sol	time (sec)
divider	7.82x	25.3x	13.2%	7.87x	76.8x	13.2%	9.14x	37.9x	55.3%	7.75x	13.2%	-	-	441
mrisc_core	21.5x	1.45x	100%	22.9x	4.59x	100%	5.23x	2.29x	100%	15.6x	100%	4.71x	100%	79.7
spi	3.24x	1.10x	100%	3.72x	3.78x	100%	2.08x	2.05x	100%	0.35x	100%	0.33x	100%	6.47
usb_core	95.1x	5.99x	100%	100x	18.5x	100%	19.9x	8.90x	100%	65.8x	100%	18.7x	100%	172
wb	4.15x	0.60x	100%	5.12x	2.70x	100%	3.45x	1.61x	100%	1.12x	100%	0.97x	100%	3.05
sudoku	-	2.02x	100%	-	6.11x	100%	7.59x	3.30x	100%	-	-	-	-	1043
or1200	-	23.6x	23.1%	5.50x	102x	23.1%	1.19x	23.7x	100%	-	-	-	-	42260
shift1add256	919x	141x	6.78%	1376x	492x	6.78%	650x	296x	11.9%	12.4x	67.8%	9.59x	94.9%	109
shift1add512	3540x	688x	6.78%	6023x	2366x	6.78%	3275x	1474x	11.9%	25.0x	67.8%	17.2x	94.9%	564
cmugigamax	4.48x	0.35x	94.0%	9.03x	3.42x	94.0%	5.87x	0.80x	100%	3.58x	94.0%	2.30x	100%	15.6
bjrb07amba1	71.5x	34.9x	12.5%	104x	120x	12.5%	52.1x	49.5x	80.0%	88.3x	12.5%	17.0x	80.0%	181
bobuns2p10	1.74x	0.04x	100%	1.74x	0.11x	100%	0.47x	0.10x	100%	1.72x	100%	-	-	576
AVERAGE	28.8x	5.15x	63.0%	31.0x	19.1x	63.0%	12.8x	9.2x	79.9%	7.45x	75.5%	4.5x	96.2%	
MEDIAN	14.7x	4.01x	97.0%	9.03x	12.3x	97.0%	6.73x	6.1x	100%	10.1x	97.0%	7.2x	100%	

the name of the benchmark. The next nine columns show the speedup of the proposed approach relative to the exact approach, the speedup of the equivalent BMC-like approach relative to the exact approach, and the fraction of solutions found for variants of the approximate approach. The next four columns show the same information for Algorithm 2, omitting the equivalent BMC speedup as no BMC-like equivalent exists for this algorithm. The last column shows the runtime of the exact approach without the optimization discussed in Section IV-D. That optimization is evaluated separately later in this section. The number of solutions for the exact approach is omitted as it finds every solution. The "average" row shows the geometric mean of the speedup columns (as it can range from 0 to infinity) and arithmetic mean of the fraction of solutions found columns (as it has a fixed range from 0% to 100%).

The results in Tables I and II clearly demonstrate the effectiveness of the presented algorithms and the tradeoffs between them. It can be seen that the entire solution set to the problem tends to be a small portion of the design locations, with a median of 1.55% and an average of 16.5% of the design locations being solutions. In six cases, the fastest approximate configuration tested ($K = 20, N = 1$) is adequate to find the entire solution set. In every benchmark, the approximate approach with $K = \infty$ and $N = 5$ is able to find at least 80% of the solution set. It can also be seen that the proposed approximate approaches compare favorably against a BMC-like equivalent, especially for large values of K.

Fig. 7 visualizes the number of solutions found for the fastest approximate configuration tested and the exact approach. It can be seen that the approximate approach effectively presents a configurable tradeoff between runtime and number of solutions found (i.e., resolution). In the fastest configuration tested, it provides an average speedup of 31× while finding an arithmetic mean of 63% of the solution set. Increasing the window size to 5 reduces the speedup to 12.8×, but increases the fraction of solutions found to 79.9%. Note that in this configuration, K was accordingly reduced to 16 to keep $K + N$ constant between the two configurations. The unlimited cycle unreachability approach of Section III-C finds

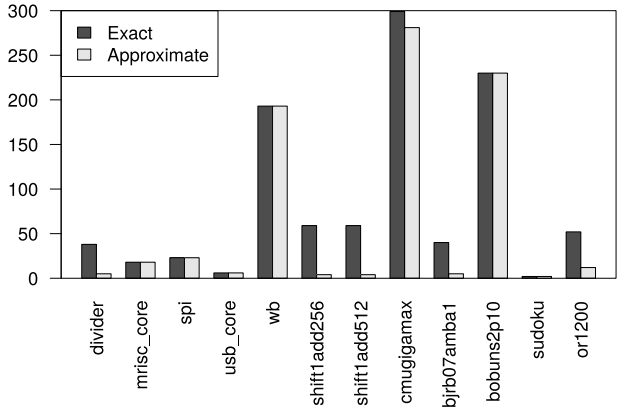

Fig. 7. Solutions found by E and A ($K = 20$ and $N = 1$) approaches.

an even larger portion of the solution set, but requires even more runtime. Naturally, the exact approach finds the full solution set but requires the greatest runtime. It additionally confers greater confidence in its results, as the user can be assured that the true error source is in the solution set.

A. Approximate Approach

It can be seen that in many cases, the approximate approach is adequate to find the complete solution set to the problem. For the `divider` benchmark, however, the window size parameter seems to have a profound impact on the number of solutions found. Fig. 8 plots the number of solutions against the window size for `divider`. Recall, from Table I the total number of solutions is 38. It can be seen that a small number of solutions are found for small N, and the number gradually increases with increasing values of N. Finally, it plateaus at $N = 10$. This is because the error occurs in a pipelined portion of the design, which has eight stages. When $N = 1$, the algorithm is only able to find locations, where the effect of the change propagates to the registers that define the target state within one cycle. In other words, it can only find solutions in the one-step cone-of-influence of the target state. In the `divider` benchmark, this is essentially the output stage

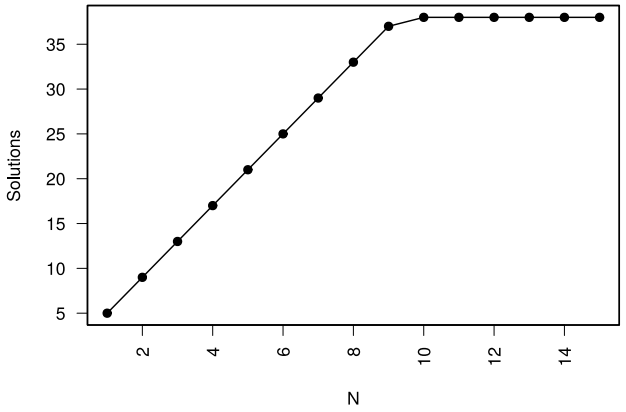

Fig. 8. Solutions versus window size for `divider` ($K = 20$).

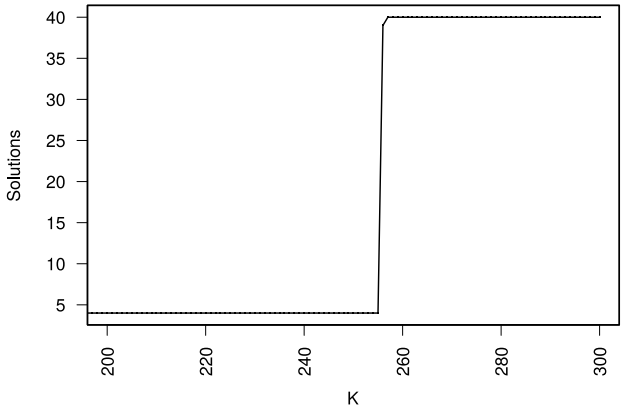

Fig. 9. Solutions versus cycle limit for `shift1add256` ($N = 1$).

of the pipeline. Increasing N allows solutions to be found in other pipeline stages. This matches the observed results, as four additional solutions are found each time N is increased, until it plateaus after $N = 9$.

The cycle limit parameter also impacts the number of solutions found in many cases. Fig. 9 plots the number of solutions found versus the cycle limit for `shift1add256`, which has a total of 59 solutions. It can be seen that the majority of solutions are found with very high values of K. In particular, when increasing K from 255 to 256, 35 additional solutions are found. One further solution is found by increasing K from 256 to 257. As many solutions only appear at high values of K, it can be inferred that some states are only reachable in a large number of cycles for this circuit, and that many states relevant to this problem can only be reached after 256 cycles. Further, no value of K is large enough to find all solutions when $N = 1$, as evidenced by the fact that the unlimited cycle approximate approach finds only 40 out of 59 total solutions when $N = 1$.

Naturally, adjusting these parameters can also impact runtime. Fig. 10 plots the number of solutions found and runtime against N for the `bjrb07amba1` benchmark. Fig. 11 plots those metrics against K for `shift1add512`. Each of these benchmarks has substantial variation in the number of solutions when varying the respective parameter (K or N). It can

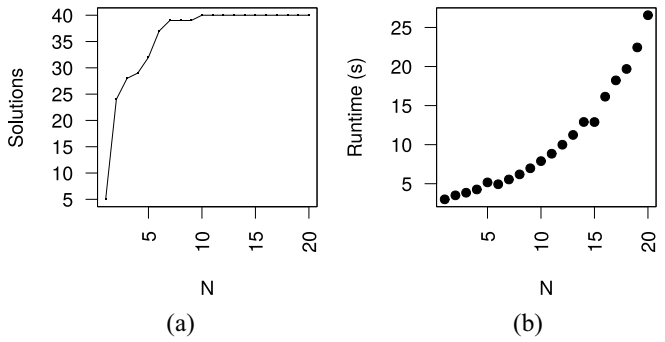

Fig. 10. Solutions and runtime versus N for `bjrb07amba1` ($K = 100$).

be seen that the runtime appears to scale somewhat similarly to the number of solutions found as K increases, with both metrics increasing sharply around $K = 512$. When considering N, runtime seems to scale up as N increases. The upward trend in runtime with increasing N is intuitive, as the SAT-based debugging steps become more complex when more time-frames are added.

However, the close tracking with the number of solutions as K increases appears less intuitive. This occurs because the presence of solutions tends to lead to several reachability checks. Obviously a solution's presence in L requires at least one reachability check to verify that it is a solution. In practice, it tends to be the case that there are multiple F_K-states that can transition to the target state if a solution's corresponding error-select register is activated. Some of those states may not be K-step reachable, and which one the SAT solver chooses is essentially random. If the SAT solver chooses a non-K-step reachable state, the suspect is discarded as a spurious solution. This can happen several times before a K-step reachable state is chosen, resulting in a suspect being detected as spurious several times before finally being proven as a real solution.

Table III demonstrates this phenomenon. The first column shows the name of the benchmark. The next two show the values for K and N, respectively. Columns 3 and 4 contain the number of spurious solutions discarded and the number of spurious solutions discarded that were not later found to be real solutions. The final column shows the number of distinct locations that were found to be spurious solutions. That is, if a location is found to be a spurious solution multiple times, it is only counted once. It can be seen that in many cases, a large majority of the solutions detected as spurious were later found to be real solutions. This demonstrates how the presence of solutions in L can lead to more reachability checks than might be expected. Additionally, from the small number of unique spurious solutions, it can be seen that in many cases, a small number of troublesome locations give rise to a large number of spurious solutions, and therefore reachability checks.

B. Exact Approach

As it returns the complete solution set, the exact approach has no parameters that allow it to tradeoff runtime versus the number of solutions found. However, two key performance

TABLE III
SPURIOUS SOLUTIONS FOUND FOR SELECTED BENCHMARKS

benchmark	K	N	total spurious	true spurious	unique spurious
cmugigamax	16	5	13	10	10
cmugigamax	20	1	10	1	7
cmugigamax	50	1	9	1	6
cmugigamax	∞	1	17	2	13
cmugigamax	∞	5	23	8	14
wb	16	5	35	35	4
wb	20	1	40	19	4
wb	50	1	40	19	4
wb	∞	1	38	16	4
wb	∞	5	38	16	4
spi	16	5	0	0	0
spi	20	1	20	0	3
spi	50	1	14	0	2
spi	∞	1	50	0	4
spi	∞	5	50	0	4

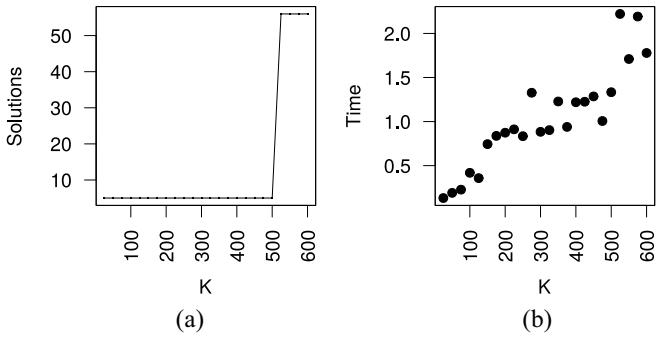

Fig. 11. Solutions and runtime versus K for `shift1add512` ($N = 2$).

TABLE IV
COMPARISON OF EXACT APPROACH OPTIMIZATIONS

		Incr.	**Pruning**		**Non-Incremental**	
benchmark	proof level	time (s)	time (s)	speedup	time (s)	speedup
divider	1	441	445	0.99x	7403	0.060x
mrisc_core	2	78.7	79.0	1.00x	97.6	0.81x
spi	1	6.47	5.19	1.25x	526	0.012x
usb_core	1	172	230	0.75x	196	0.878x
wb	2	3.05	3.25	0.94x	1636	0.002x
sudoku	1	1043	2350	0.44x	18750	0.056x
or1200	8	42260	-	-	-	-
shift1add256	79	109	80.0	1.36x	22260	0.005x
shift1add512	140	564	348	1.62x	-	-
cmugigamax	1	15.6	17.2	0.91x	14780	0.001x
bjrb07amba1	9	181	224.7	0.81x	2442	0.074x
bobuns2p10	1	576	553	1.04x	713	0.808x
AVERAGE				0.96x		0.042x

optimizations are proposed in this paper. Section IV-C proposes the use of incrementality and Section IV-D proposes a performance optimization, where bad lemmas are pruned. Table II displays runtime for the incremental approach without bad clause pruning. This is because incrementality is consistently a performance gain, whereas bad lemma pruning introduces extra overhead and in some cases can result in increased runtime.

Table IV compares the runtime of the exact approach with various optimizations turned on. The first column shows the name of the benchmark. The second shows the level at which a proof is discovered during model checking using PDR. The next six show the runtime and speedup for the incremental approach, the incremental approach with bad clause pruning, and the nonincremental approach without pruning, respectively. It can be seen that the pruning causes substantial speedups in some cases (e.g., `shift1add256`) and slowdowns in other cases (e.g., `sudoku`).

As mentioned in Section IV-D, it appears to be effective in cases, where a long inductive trace is needed, such as the `shift1add256` benchmark. With a long inductive trace, not pushing bad clauses forward saves substantial runtime. While the optimization appears to be effective in limited situations, problems involving long inductive traces can be particularly expensive. The optimization may have substantial value in these cases. As we expect our approaches would be applied after model checking to prove the target state unreachable, the level at which a proof is discovered could provide a rough proxy for the applicability of this optimization. If model checking requires a long inductive trace, we expect our debugging approaches to require the same, providing guidance as to whether or not to apply this optimization.

On the other hand, incrementality is a consistent performance benefit. While it does introduce some overhead as there are potentially more clauses in the inductive trace, in all cases this is more than compensated for by the saved effort. Evidently, many of the clauses that were learned in earlier calls to PDR remain relevant for later calls. Incrementality appears to offer a greater benefit to problem instances with more solutions (e.g., `wb`). This is as expected, as in the nonincremental version of the approach, each solution results in a new call to PDR. Without incrementality, each call starts with no inductive trace, and therefore must learn all new clauses. In practice, since very little changes are made to the model between subsequent calls, it is expected that many of the clauses from the previous run must be relearned, resulting in significant repetition. Additionally, the presence of clauses from previous runs increases the accuracy of the inductive trace, and allows PDR to learn "better" clauses. This occurs because fewer unreachable states satisfy each F_i, and therefore generalization is more effective. Across all circuits, incrementality offers a 23.9x speedup.

VI. CONCLUSION

Modern verification environments often seek to verify liveness properties in addition to safety properties. While automation is available to aid in debugging failed safety properties, debugging failed liveness properties is a predominantly manual task. This paper presents methodologies to diagnose erroneously unreachable states. The first is an approximate approach that uses steps of state space approximation, SAT-based debugging, and spurious solution detection to find a subset of all solutions to the problem. The second is an exact approach that solves unbounded model checking problems using an enhanced FSM model of the circuit to find all solutions to the problem by making extensive use of incremental PDR. Experiments are presented demonstrating the tradeoffs between the presented approaches. The exact approach provides a means of finding the entire solution set to the problem, but may require substantial runtime.

The approximate approach provides the user with a configurable tradeoff between runtime and resolution. The experiments further demonstrate the substantial benefits achieved by applying PDR in an incremental fashion.

REFERENCES

[1] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2015, pp. 1–6.

[2] A. R. Bradley, "Sat-based model checking without unrolling," in *Proc. Int. Conf. Verification Model Checking Abstract Interpretation*, Austin, TX, USA, 2011, pp. 70–87.

[3] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic verification of arithmetic circuits in RTL using stepwise refinement of term rewriting systems," *IEEE Trans. Comput.*, vol. 56, no. 10, pp. 1401–1414, Oct. 2007.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," in *Advances in Computers*, vol. 58. Cambridge, MA, USA: Academic Press, 2003, pp. 118–149.

[5] K. L. McMillan, "Interpolation and sat-based model checking," in *Proc. Comput. Aided Verification (CAV)*, Boulder, CO, USA, 2003, pp. 1–13.

[6] A. Ivrii and A. Gurfinkel, "Pushing to the top," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Austin, TX, USA, 2015, pp. 65–72.

[7] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, Austin, TX, USA, 2011, pp. 125–134.

[8] E. Ábrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde, "Parallel SAT solving in bounded model checking," in *Formal Methods: Applications and Technology* (LNCS 4346). Heidelberg, Germany: Springer, 2007, pp. 301–315.

[9] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu, "Efficient modular SAT solving for IC3," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Portland, OR, USA, Oct. 2013, pp. 149–156.

[10] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

[11] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1597–1608, Oct. 2009.

[12] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 11, pp. 1790–1803, Nov. 2010.

[13] S. Safarpour, A. Veneris, and F. Najm, "Managing verification error traces with bounded model debugging," in *Proc. 15th Asia South Pac. Design Autom. Conf. (ASP DAC)*, Taipei, Taiwan, 2010, pp. 601–606.

[14] H. Mangassarian, B. Le, and A. Veneris, "Debugging RTL using structural dominance," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*, vol. 33, no. 1, pp. 153–166, Jan. 2014.

[15] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Cambridge, U.K., Oct. 2012, pp. 52–59.

[16] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 6, pp. 1138–1149, Jun. 2008.

[17] R. K. Ranjan, C. Coelho, and S. Skalberg, "Beyond verification: Leveraging formal for debugging," in *Proc. 46th ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jul. 2009, pp. 648–651.

[18] R. Berryhill and A. Veneris, "Automated rectification methodologies to functional state-space unreachability," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Grenoble, France, 2015, pp. 1401–1406.

[19] R. Berryhill and A. Veneris, "Diagnosing unreachable states using property directed reachability," in *Proc. Workshop Constraints Formal Verification (CFV)*, 2015. [Online]. Available: http://www.eecg.utoronto.ca/~veneris/2cfv15.pdf

[20] R. Berryhill and A. Veneris, "A complete approach to unreachable state diagnosability via property directed reachability," in *Proc. Asia South Pac. Design Autom. Conf. (ASP DAC)*, 2016, pp. 127–132.

[21] (2007). *OpenCores.org*. [Online]. Available: http://www.opencores.org

[22] H. Mangassarian, A. Veneris, and M. Benedetti, "Robust QBF encodings for sequential circuits with applications to verification, debug, and test," *IEEE Trans. Comput.*, vol. 59, no. 7, pp. 981–994, Jul. 2010.

[23] P.-Y. Chung and I. N. Hajj, "Diagnosis and correction of multiple logic design errors in digital circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 5, no. 2, pp. 233–237, Jun. 1997.

[24] K. Gitina *et al.*, "Equivalence checking of partial designs using dependency quantified Boolean formulae," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Asheville, NC, USA, Oct. 2013, pp. 396–403.

[25] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with IC3," in *Proc. Formal Methods Comput.-Aided Design*, Portland, OR, USA, Oct. 2013, pp. 165–168.

[26] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *Proc. 15th Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, Trento, Italy, 2012, pp. 157–171.

[27] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, Austin, TX, USA, 2011, pp. 135–143.

[28] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002.

Ryan Berryhill (S'15) received the B.A.Sc. degree in computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 2014, and the M.A.Sc. degree in computer engineering from the University of Toronto, Toronto, ON, USA, in 2016, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering.

His current research interests include inductive formal verification and automated formal debugging of digital systems.

Andreas Veneris (S'96–M'99–SM'05) received the Diploma degree in computer engineering and informatics from the University of Patras, Patras, Greece, in 1991, the M.S. degree in computer science from the University of Southern California, Los Angeles, CA, USA, in 1992, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1998.

In 1998, he was a Visiting Faculty Member with the University of Illinois at Urbana–Champaign, until 1999, when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science with the University of Toronto, Toronto, ON, USA, where he is a Professor. He has authored one book and he holds several patents. His current research interests include CAD for debugging, verification, synthesis, and test of digital circuits/systems, combinatorics, and leger-based technologies.

Dr. Veneris was a recipient of several teaching awards and a Best Paper Award. He is a member of ACM, AMS, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.